ABSTRACT

POOJARY, VIKRAM S. Predicting Loop Unrolling Impact in OpenMP Programs Using Machine Learning . (Under the direction of Dr. Yan Solihin).

Performance tuning of high performance numerical code is an important process which is still largely performed manually. While recent research in automated performance tuning has proposed run-time application configuration and compilation, most compilers in use today do not support such run-time features. As a result, a performance tuner's role is limited to selecting the right compiler optimizations for a given application and environment in which the application runs. Because many compiler optimizations do not give performance benefits in all cases, performance tuners must tediously test each optimization on their applications under a wide range of scenarios. Therefore, it is desirable to automate compiler optimization selection in order to avoid or at least reduce the tuning effort.

This thesis deals with the question of whether machine learning techniques can be used to automate compiler optimization selection. It presents a case study in which an Artificial Neural Network (ANN) and a Decision Tree (DT) are constructed, trained, and used to predict whether, for a given loop nest in a shared memory parallel program, loop unrolling optimization should be applied or not. Simple characteristics of the loop nests, such as the nesting level, iteration count, and body size, are collected and used as input to the ANN or DT. The ANN and DT were trained with loop nests from some OpenMP-based NAS parallel benchmarks, and are used to predict the benefit of loop unrolling across different benchmarks, and across different numbers of parallel threads.

Various training methods were tried, and in the best case, ANN predicts correctly whether loop unrolling is beneficial in 62% of the cases, whereas DT predicts correctly whether loop unrolling is beneficial in 56% of the cases. Although the results show promise, we believe that to accurately automate compiler optimization selection, many other factors may need to be taken into account in characterizing each loop nest, due to complex interactions of loop unrolling with memory hierarchy, data layout, thread partitioning, and instruction-level parallelism.

Predicting Loop Unrolling Impact in OpenMP Programs Using Machine Learning

by

Vikram S. Poojary

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh

2005

Approved By:

Dr. Gregory Byrd	Dr. Edward Gehringer

Dr. Yan Solihin
Chair of Advisory Committee

Dedicated to my family.

Biography

Vikram Poojary was born on 27th November, 1978, in Mumbai, India. He graduated from Shah and Anchor Kutchhi College of Engineering, Mumbai, with a Bachelor of Engineering degree in Computer Engineering in August 2000. From October 2000 to July 2002, he worked as Systems Engineer at Wipro Technologies, Hyderabad, India. He enrolled in the graduate program in Computer Science at North Carolina State University in Fall 2002. With the defense of this thesis, he is receiving the Master of Science in Computer Science degree.

Acknowledgements

I would like to take this opportunity to thank my advisor Dr. Solihin for his guidance and patience on this project. I would also like to thank Dr. Byrd and Dr. Gehringer for agreeing to be on my committee. I would like to thank Dr. Semazzi of Marine, Earth and Atmospheric Sciences department for providing support for a significant duration of my Master's program. Last but not the least, I would like to thank my parents and brother for their constant love and support, without which none of this would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance tuning of high performance numerical code is an important process which is still largely performed manually. Recent research in automated performance tuning has proposed run-time application configuration and compilation. For example, ATLAS [20] and Harmony [19] have tried to automatically select the best library that improves the overall program performance. Autopilot [16, 15] and AppLeS [1] have tried to automatically adjust run-time parameters to improve performance. In the context of shared memory architectures, there are limited forms of automated tuning, such as user-level page allocation [13] and loop iteration distribution [12].

Unfortunately, most compilers in use today do not support such run-time tuning features. This may be caused partly by the overheads incurred by run-time tuning, or the assumption that when compiled with the right optimizations, applications will perform well without needing run-time tuning. As a result, other than modifying the algorithm or application source code, in many cases, the performance tuner's role is limited to selecting the right compiler optimizations for a given application and environment in which the application runs. Because many compiler optimizations do not give performance benefits in all cases, the performance tuner must tediously test each optimization on their applications under a wide range of scenarios. Therefore, it is desirable to automate compiler optimization selection in order to avoid or at least reduce the tuning effort.

The difficulty of performance tuning arises when the benefit from a particular performance optimization is not only specific to the program, but also specific to the different

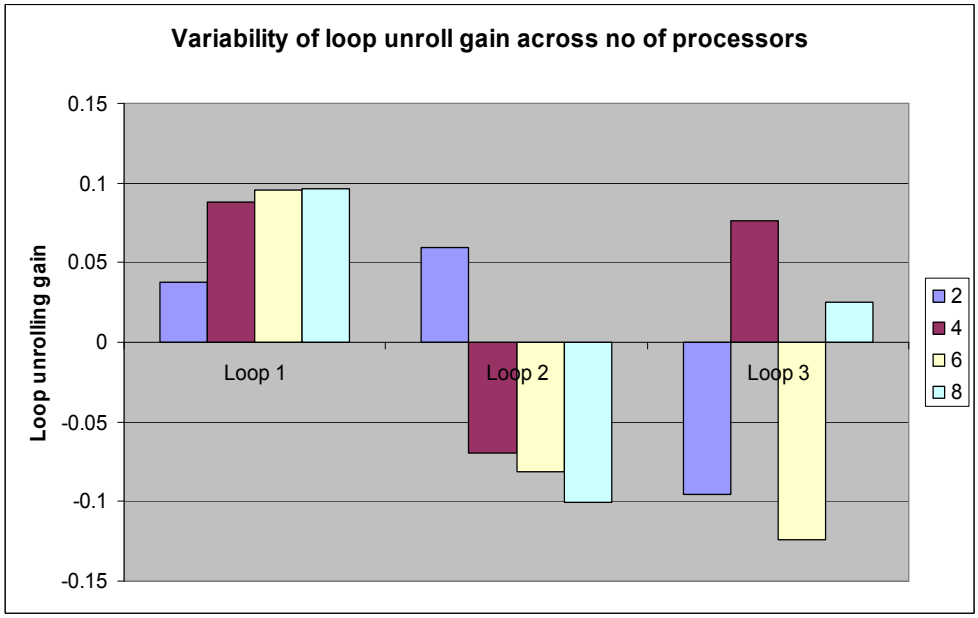


Figure 1.1: Unroll gain of three selected loops from BT benchmark across different number of parallel processors.

code sections in a program, number of parallel threads, and input sets of the program. To illustrate the extent of such dependence, Figure 1.1 and 1.2 show the variability of "unroll gain" for selected loops from BT benchmark, across different number of processors and matrix sizes, respectively. The *unroll gain* is related to the speedup produced by loop unrolling optimization, i.e.:

$$Unroll\ gain = \frac{Execution\ time\ without\ unroll - Execution\ time\ with\ unroll}{Execution\ time\ without\ unroll} \quad (1.1)$$

Figure 1.1 shows that for *Loop 1*, the unroll gains on 4, 6, and 8 threads are significantly higher than on 2 threads. From tuning perspective, this does not cause a problem because loop unrolling is still beneficial across different number of threads. However, *Loop 2* and *Loop 3* present a serious problem for performance tuning, because for some number of threads, the unroll gain is positive, while for others, the unroll gain is negative. In this case, unless the performance tuners correctly figure out the number of threads that will be used, the tuned application may not perform well.

Figure 1.2 shows two loops from the BT benchmark for seven different matrix sizes. The matrices used by BT's main computation are three dimensional, and the number of elements per dimension can be specified through the input file. The figure shows that for

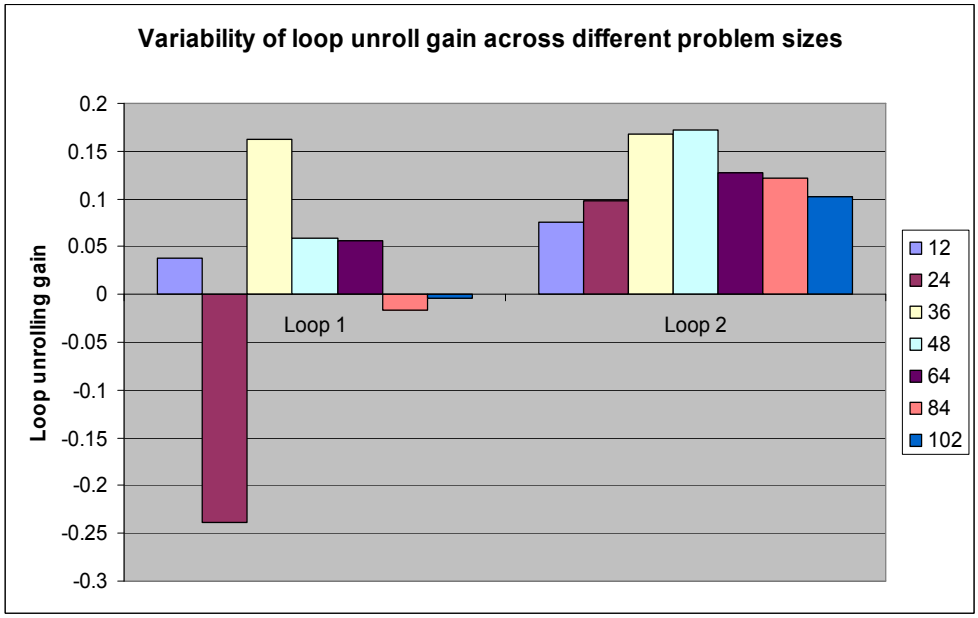


Figure 1.2: Unroll gain of three selected loops from BT benchmark across different number of elements per dimension of the main matrix structures.

Loop 1, the matrix size has a large effect on the unroll gain, which can be as low as -23%, and can be as high as 17%.

This thesis deals with the question of whether machine learning techniques can be used to automate compiler optimization selection. It presents a case study in which an Artificial Neural Network (ANN) and a Decision Tree (DT) are constructed, trained, and used to predict whether for a given loop nest in a shared memory parallel program, loop unrolling optimization should be applied or not.

ANN or DT needs to be able to correlate the application code characteristics with the loop unrolling effectiveness. Such code characteristic parameters must not be beyond a few parameters that can be easily obtained with static code analysis and that are typically available to performance tuners. Other schemes that require run-time parameter extraction, as well as complex static code analysis are not preferable because they may add a significant complexity to the compiler and the run-time system, may add significant performance overheads, and are typically not available to performance tuners. Therefore, simple characteristics of the loop nests, such as the nesting level, iteration count, and body size, are collected and used as input to the ANN or DT. The ANN and DT were trained with loop nests from some OpenMP-based NAS parallel benchmarks, and are used to predict

the benefit of loop unrolling across different benchmarks, and across different number of parallel threads.

To test the promise of the ANN or DT approach, we perform our experiments with loop unrolling flag on OpenMP NAS benchmarks with the IBM OpenMP Fortran compiler `xlf_r` on a 32-processor IBM p690 shared memory multiprocessor system running the AIX operating system version 5.1. We use the `-nounroll` compiler flag to selectively enable and disable unrolling for loops in a single source file.

Various training methods were tried. For interpolated data points, the ANN and DT are able to predict whether loop unrolling is beneficial for different number of threads of the same application with 71.5% and 70.33% accuracy in the best case evaluated, respectively. For extrapolated data points, the ANN and DT are able to predict whether loop unrolling is beneficial for different number of threads of the same application with 66.77% and 70.67% accuracy in the best evaluated case, respectively. Finally, predicting across benchmarks results in lower accuracy. ANN predicts correctly whether loop unrolling is beneficial in 62% of the cases, whereas DT predicts correctly whether loop unrolling is beneficial in 56% of the cases. Although the results show promise, we believe that to accurately automate compiler optimization selection, many other factors may need to be taken into account in characterizing each loop nest due to the complex interactions of loop unrolling with memory hierarchy, data layout, thread partitioning, and instruction-level parallelism.

The organization of the thesis is as follows. Chapters 2, 3 and 4 elaborate upon the OpenMP programming model, loop unrolling, and the machine learning techniques of neural networks and decision trees respectively. Chapter 5 discusses the experimental setup and methodology. Chapter 6 presents the results of the experiments. Chapter 7 discusses related work. Chapter 8 presents the conclusions.

Chapter 2

OpenMP parallel programming paradigm

OpenMP is a portable standard for shared memory parallel programming [4]. It provides a set of directives and runtime library routines in Fortran and C/C++ to write parallel applications, whenever the shared memory abstraction is available. It follows a fork-join model. The sequential portion of the code is executed by a master thread. On encountering a parallel region (for example, when directives such as "PARALLEL" and "END PARALLEL" are encountered), slave threads are created. The master thread and the slave threads execute the section of code enclosed within the parallel region. At the end of the parallel region, the threads synchronize and the master thread continues execution. Work-sharing constructs are provided, which distribute the work in a loop between the parallel threads (by means of directives such as "PARALLEL DO"). Synchronization directives, such as barriers and critical sections, are also available in this standard. An implied synchronization occurs at the end of the "DO" loop unless an "END DO NOWAIT" is specified. Data sharing of variables is specified at the start of parallel or worksharing constructs, using the SHARED and PRIVATE clauses. In addition, reduction operations(such as summation) for the scalar variables are specified by the "REDUCTION" clause. Figure 2.1 illustrates a code fragment that shows OpenMP in action.

```
!$omp parallel default(shared) private(i,j,k,m,rho_inv,uijk,up1,um1,
!$omp&   vijk,vp1,vm1,wijk,wp1,wm1)
c---------------------------------------------------------------------
c     compute the reciprocal of density, and the kinetic energy,
c     and the speed of sound.
c---------------------------------------------------------------------
!$omp do
      do k = 0, grid_points(3)-1
         do j = 0, grid_points(2)-1
            do i = 0, grid_points(1)-1
               rho_inv = 1.0d0/u(1,i,j,k)
               rho_i(i,j,k) = rho_inv
               us(i,j,k) = u(2,i,j,k) * rho_inv
               vs(i,j,k) = u(3,i,j,k) * rho_inv
               ws(i,j,k) = u(4,i,j,k) * rho_inv
               square(i,j,k)     = 0.5d0* (
     >                 u(2,i,j,k)*u(2,i,j,k) +
     >                 u(3,i,j,k)*u(3,i,j,k) +
     >                 u(4,i,j,k)*u(4,i,j,k) ) * rho_inv
               qs(i,j,k) = square(i,j,k) * rho_inv
            enddo
         enddo
      enddo
!$omp end do nowait

c---------------------------------------------------------------------
c copy the exact forcing term to the right hand side;  because
c this forcing term is known, we can store it on the whole grid
c including the boundary
c---------------------------------------------------------------------

!$omp do
      do k = 0, grid_points(3)-1
         do j = 0, grid_points(2)-1
            do i = 0, grid_points(1)-1
              do m = 1, 5
                  rhs(m,i,j,k) = forcing(m,i,j,k)
               enddo
            enddo
         enddo
      enddo
!$omp end do
```

Figure 2.1: OpenMP code fragment from benchmark BT

OpenMP provides "orphan" directives that are encountered outside the lexical extent of parallel regions. It allows the user to specify control from anywhere inside the parallel region and aids greatly in implementing coarse grain parallel algorithms. Figure 2.2 illlustrates a code fragment that shows orphan directives.

```
c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

      subroutine blts ( ldmx, ldmy, ldmz,
     >                  nx, ny, nz, k,
     >                  omega,
     >                  v, tv,
     >                  ldz, ldy, ldx, d,
     >                  ist, iend, jst, jend,
     >                  nx0, ny0 )

c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
c
c   compute the regular-sparse, block lower triangular solution:
c
c                     v <-- ( L-inv ) * v
c
c-----------------------------------------------------------------------

      implicit none

c-----------------------------------------------------------------------
c  input parameters
c-----------------------------------------------------------------------
      integer ldmx, ldmy, ldmz
      integer nx, ny, nz
      integer k
      double precision  omega
c-----------------------------------------------------------------------
c   To improve cache performance, second two dimensions padded by 1
c   for even number sizes only.  Only needed in v.
c-----------------------------------------------------------------------
      double precision  v( 5, ldmx/2*2+1, ldmy/2*2+1, *),
     >        tv( 5, ldmx/2*2+1, ldmy),
     >        ldz( 5, 5, ldmx/2*2+1, ldmy),
     >        ldy( 5, 5, ldmx/2*2+1, ldmy),
     >        ldx( 5, 5, ldmx/2*2+1, ldmy),
     >        d( 5, 5, ldmx/2*2+1, ldmy)
      integer ist, iend
      integer jst, jend
      integer nx0, ny0

c-----------------------------------------------------------------------
c  local variables
c-----------------------------------------------------------------------
      integer i, j, m
      double precision  tmp, tmp1
      double precision  tmat(5,5)
```

```
c---------------------------------------------------------------------
c  Thread synchronization for pipeline operation
c---------------------------------------------------------------------
      include 'npbparams.h'
      integer isync(0:isiz2), mthreadnum, iam, neigh
      common /threadinfo/ isync, mthreadnum
      external omp_get_thread_num
      integer omp_get_thread_num


!$omp do schedule(static)
      do j = jst, jend
         do i = ist, iend
            do m = 1, 5

                  tv( m, i, j ) =  v( m, i, j, k )
     >    - omega * (  ldz( m, 1, i, j ) * v( 1, i, j, k-1 )
     >               + ldz( m, 2, i, j ) * v( 2, i, j, k-1 )
     >               + ldz( m, 3, i, j ) * v( 3, i, j, k-1 )
     >               + ldz( m, 4, i, j ) * v( 4, i, j, k-1 )
     >               + ldz( m, 5, i, j ) * v( 5, i, j, k-1 )  )

            end do
         end do
      end do
!$omp end do nowait
```

Figure 2.2: OpenMP code fragment from benchmark LU illustrating orphan directives

Chapter 3

Loop Unrolling Optimization

Loop unrolling is a well known and widely used compiler technique that simply replicates the loop body multiple times, adjusting the loop termination code [6]. It exposes instruction level parallelism by increasing the number of instructions relative to the branch and overhead instructions. Since the loop termination test and backward branch is encountered only at the end of the unrolled loop body, extra test and branch operations can be removed. Loop unrolling can often result in better scheduling, as unrolling the loop exposes more independent computations that can be scheduled to minimize stalls. It is especially useful in optimizations targeting memory instructions, such as rescheduling to exploit memory locality, scalar replacement of memory references to the same location (on consecutive iterations) etc. Loop unrolling also exposes adjacent memory references which can then be merged into a single wide reference.

While, theoretically, loop unrolling is a beneficial optimization, gains from loop unrolling can be limited by secondary effects. One factor is the growth in code size which can result in instruction cache performance degradation. A more important limitation arises from increased register pressure, as the number of live values increases with unrolling. In systems with long memory latencies, register spills and reload costs may override the benefits from unrolling. As the impact of loop unrolling is mostly realized in secondary effects, it is difficult to predict when loop unrolling will be beneficial.

Loop unrolling performance is affected by many aspects of the system, including the instruction scheduler, register architecture, memory system and underlying architecture.

Depending on the execution context and interaction of system parameters, loop unrolling may improve or hurt overall performance. Under such circumstances, the only way to make informed unrolling decisions is by using real world empirical data, and machine learning algorithms are a feasible and potentially superior option to utilize it, in lieu of constructing analytical models based on heuristics.

Chapter 4

Machine Learning

Machine learning techniques offer an automatic, flexible and adaptive framework for dealing with many of the parameters that decide the effectiveness of program optimizations. A database of examples is created that records the behavior of optimizations when the hypothesized features that impact an optimization are recorded along with the effectiveness of the optimization. The machine learning algorithm then attempts to learn the rules or heuristics automatically from the accumulated data. This process is impacted by the size of the database and the nature and number of the features used. We consider two such techniques in the following sections that have been widely applied in a number of different problem domains to achieve the same end result of automatically learning heuristics or rules from a recorded database.

4.1 Neural Networks

A feedforward neural network (Figure 4.1) consists of highly interconnected processing elements called nodes [17]. An interconnection between a pair of nodes has a real value associated with it called a weight. Nodes are arranged in groups called layers. There are three kinds of layers: an input layer where inputs from the external world are accepted, an output layer where the output of the neural network to the outside world is generated, and zero or more hidden layers that do all the necessary computation and collectively contribute

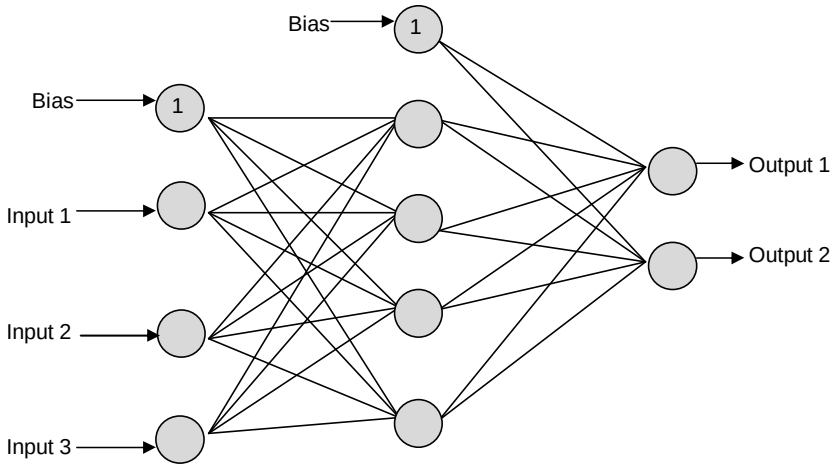


Figure 4.1: A fully connected feed-forward neural network with a 3-4-2 topology consisting of 3 input nodes, 1 hidden layer with 4 nodes and 2 output nodes. Note that a constant bias of 1 is provided to all the nodes in the hidden layer as well as output layer.

to the output of the output layer.

Each node computes a predefined function of its inputs called the "activation" function. Typically the activation function is a non-linear function such as sigmoidal or tanh. For example, if we use the tanh function as the activation function, the activity of the hidden node i, denoted by y_i, is computed as:

$$y_i = tanh(\sum_j w_{ij} x_j + b_i) \tag{4.1}$$

where x_j is the activation of input node j, w_{ij} is the connection weight from node j to node i, b_i is a bias weight associated with the node, and tanh is the hyperbolic tangent function, computed as:

$$tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}} \tag{4.2}$$

The logistic activation function is computed as:

$$logistic(u) = \frac{1}{1 + e^{-u}} \tag{4.3}$$

The behavior of a feedforward neural network is completely determined by the number of nodes in the hidden layers and the connection weights between the nodes throughout. The problem of learning in neural networks is thus one of arriving at the right set of

connection weights. One learning mechanism employed to achieve this is called backpropagation.

We start out with a fully connected feedforward neural network connected by a random set of connection weights. The network adjusts its weights based on the delta learning rule each time it sees an input-output data pair. Each pair of data goes through two stages of activation: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activation flow until the output layer is reached. During the backward pass, the network's actual output is compared with the target output and errors are computed as the difference in those outputs.

Finally, errors are propagated back to the connections stemming from the input units, adjusting the connection weights in the process. Many different error measures (also called error functions, objective functions, cost functions, or loss functions) can be used as a measure of the performance of the network. The **sum of squared errors** function:

$$E = \sum_{k} (t_j - y_j)^2 \tag{4.4}$$

where j is an index over examples in the training set, y_j is the actual output of the network when training input j is presented, and t_j is the target output-the output indicated for that example in the training set, is used in the work reported in this thesis.

One important parameter of the learning step is the learning rate, or η. It determines how much the connection weight should be adjusted to the direction shown by the error. We experiment with values of 0.01 and 0.1 for η.

A complete round of forward-backward passes and weight adjustments using all input-output pairs is called an epoch. We use an online approach to training so that the weights are updated after each example in the training set is presented to the network. A backpropagation network needs to "learn" from the same data set through hundreds–sometimes even thousands–epochs in order to gradually refine its connection weights.

Training continues until a stopping condition is reached. After training, the network has the ability to recall a previous training data set.

If j is a node in the output layer, the error at the node j is

$$\delta_j = (t_j - y_j)\nabla'(y_j) \tag{4.5}$$

where y_j is the activation of output node j, t_j is its target activation value, and δ_j is its error value that is propagated back.

If j is a node in the hidden layer, and if there are k nodes, $1, 2, \ldots, k$ that receive a connection from node j, the error at node j is

$$\delta_j = \Big(\sum_{i=1}^{k} w_{ij}\delta_i \Big) \nabla'(y_j) \tag{4.6}$$

where the weights $w_{1j}, w_{2j}, \ldots, w_{kj}$ belong to the connections from hidden node j to nodes $1, 2, \ldots, k$.

The backpropagation rule applied at time t is:

$$\Delta w_{ji}(t) = \eta \delta_j y_i + \beta \Delta w_{ji}(t-1) \tag{4.7}$$

where $\Delta w_{ji}(t)$ is the change in weight from node i to node j at time t, η is the learning rate and β is the momentum. We do not use the momentum in our experiments.

We label a fully connected feedforward neural network architecture as a-b-c where a represents the number of input nodes, b represents the number of nodes in the hidden layer, c represents the number of output nodes. For example, a 5-10-1 network consists of 5 input nodes, 1 hidden layer consisting of 10 nodes and an output layer consisting of 1 output node and is called a single hidden-layer network.

We subdivide the available data sets into "training", "validation", and "test" sets. Training set is the set of samples used for learning, that is to adjust the weights of the feed-forward network. Validation set is typically a fraction of the available training set that is used many times to detect convergence and stop training. Testing set is a set of examples used only to assess the performance of the neural network, and hence absolutely no information about the test set examples or the test set performance of the network is made available during the training process.

We define a training strip of length k [14] as a sequence of k epochs numbered $n+1 \ldots n+k$ where n is divisible by k. The training progress (measured in parts per thousand) measured after such a training strip is then

$$P_k(t) = 1000. \Big(\frac{\sum_{t' \epsilon t-k+1 \ldots t} E_{tr}(t')}{k.min_{t' \epsilon t-k+1 \ldots t} E_{tr}(t')} - 1 \Big) \tag{4.8}$$

that is, "how much was the average training error during the strip larger than the minimum training error during the strip?"

We experiment with and without validation set in this thesis. Training is stopped as soon as a minimum error on the training set is reached or a given number of training

epochs have been completed or the training progress goes down below a particular rate. In addition, if the validation set is used, the training is stopped if the error on the validation set rises a particular threshold above the minimum seen so far, else the training is stopped if the error on the training set rises a particular threshold above the minimum seen so far. After the training is stopped, the network with the minimum validation or training error is selected, depending on whether validation set is used or not.

4.2 Decision Trees

A decision tree is a directed acyclic graph in the form of a tree. Given a database of examples represented as feature vectors with assigned categories, we can learn decision trees from it to classify the data.

Each non-leaf node of a decision tree represents a test that examines the value(s) of one (or several) feature(s) of the data set, and a branch from the non-leaf node represents the possible value(s) of the feature(s) examined. All leaf nodes denote example classifications. The root node of the decision tree represents all the available data. Decision trees are then used to classify examples by first performing the test specified at the root node and then following the branch indicated by the result of the test down to the next level of the tree. We continue this process until we terminate in a unique path at the leaf node. The classification associated with the leaf node is assigned to the example. We can thus see that decision trees are directly interpretable by experts.

A decision tree can be learned by splitting the source dataset into subsets based on the value(s) of feature(s). This process is then repeated on each derived subset in a recursive manner. The recursion is completed when splitting is either not feasible, or a single classification value can be applied to each member of the derived subset.

The decision tree experiments reported in this thesis were conducted using the well-studied, and widely distributed, C4.5 system [21]. The decision tree implementation used was downloaded from a widely used online distribution [21]. C4.5 uses the concept of information gain to select the feature(s) to test at each non-leaf node. The information gain can be described as the effective decrease in entropy (derived from information theory) resulting from making a choice as to which feature to use and at what level.

To avoid overfitting the data, the tree resulting from the training process is

"pruned" back from the leaves using a user-defined level.

Chapter 5

Methodology and Experimental setup

Experiments were conducted with loops from the following Fortran OpenMP NAS benchmarks [9] :

1. BT: BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5×5 blocks and are solved sequentially along each dimension.

2. CG: CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.

3. FT: FT contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFTs, one for each dimension.

4. LU: LU is a simulated CFD application that uses symmetric successive over-relaxation

(SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

5. MG: MG uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement.

6. SP: SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

The input data sizes used in the experiments are shown in Table 5.1

5.1 Loop Characterization

For each benchmark, subroutines that collectively contribute 90% of the entire execution time are identified using the profiling tool gprof. Next, all the loops in the identified subroutines that have been parallelized using OpenMP are timed. For perfectly nested loops, only the innermost loop is timed and for imperfectly nested loops, multiple loops at the innermost level are timed independently.

For each loop, two sets of features were considered as shown in Tables 5.2 and 5.3. The primary difference between the two feature sets is that of considering the various floating point operations individually and in aggregation.

5.2 Platform

The OpenMP NAS benchmarks were run on a 32-processor(Power4 1.3Ghz) SMP IBM p690 running a single instance of the AIX operating system version 5.1. The IBM OpenMP compiler *xlf_r* for OpenMP Fortran was used to compile the benchmarks. The normative compiler options of *qarch=pwr4, qtune=pwr4, qcache=auto* and optimization

level of *O3* was used. The benchmarks were run in the prescence of other workloads on the machine.

5.3 Experimental Methodology

The IBM OpenMP compiler for Fortran, xlf_r, has a flag -qnounroll that can be used to suppress unrolling of loops. The flag, -qunroll, or the -O3 optimization level enables unrolling by default, but does not guarantee that the unrolling will be performed. The assembly code was checked to verify that all the timed loops were unrolled.

The various benchmarks mentioned above were executed 10 times for 2, 4, 6 and 8 threads, with the unrolling flag both enabled and disabled, as described above. Each thread was requested to be scheduled on a separate processor. Feature set A was reported for the timed loops, and Feature set B was derived from it for later calculations by aggregating the various floating point operations.

The process for collecting loop data is outlined as follows:

1. Note the values of loop features for each loop by manual inspection of the benchmark source code.

2. Run each benchmark for a given number of threads with the -qnounroll option and time the marked loops.

3. Run each benchmark for a given number of threads with the -qunroll option and time the marked loops.

4. Calculate the loop unrolling gain as in equation 1.1 for each timed loop and note it as a loop feature. If the loop unrolling gain is positive, it means that loop unrolling for that particular loop signature is beneficial, and if the loop unrolling gain is negative, the loop unrolling gain for that particular loop signature is not beneficial.

We conduct three classes of experiments on the loop data that is collected:

- *Interpolation*: We consider two cases. Given information of loops across 2,4 and 8 threads for a given benchmark, we try to predict the loop unrolling benefit for 6 threads, and denote this as Interpolation Case v1. In addition, given information of

loops across 2, 6 and 8 threads for a given benchmark, we try to predict the loop unrolling benefit for 4 threads, and denote this as Interpolation Case v2.

- *Extrapolation*: We consider two cases. Given information of loops across 2 and 4 threads for a given benchmark, we try to predict the loop unrolling benefit for 6 threads, and denote this as Extrapolation Case v1. In addition, given information of loops across 2, 4 and 6 threads for a given benchmark, we try to predict the loop unrolling benefit for 8 threads, and denote this as Extrapolation Case v2.

- *Benchmark-to-Benchmark prediction*: Given the information about loops for one benchmark, we try to predict the loop unrolling behavior for each of the remaining benchmarks under consideration, for any number of threads.

We define the "raw accuracy" statistic for all classes of experiments as follows:

$$Raw\,accuracy = \frac{Number\,of\,times\,loop\,unrolling\,benefit\,predicted\,correctly}{Total\,number\,of\,testing\,set\,elements} \tag{5.1}$$

5.3.1 Neural Networks

All three classes of experiments were conducted by utilizing the numerical value of the loop unroll gain as the feature to be predicted. The neural network results were evaluated by considering the sign of the predicted unroll gain. The neural network prediction is considered to be accurate if the predicted and actual numerical value of unroll gain is either both positive or both negative. This can be interpreted as if

Both one and two levels of hidden layers were evaluated. Initial weights of the neural network were systematically varied so as to be reproducible. The number of hidden nodes considered at each hidden layer were 5, 10, 15 and 20. Loop data with feature sets A and B were considered with and without validation sets for all classes of experiments to yield four cases for each experimental class. Whenever validation sets were used, 20% of the training examples were used for validation. For each of the three classes of experiments, the neural network with the minimum training(validation, if the validation set was used) error was considered for reporting the result.

In addition to the "raw accuracy" statistic defined by equation 5.1, a further accuracy metric using 95% confidence interval obtained by incorporating the average error of the training set is reported, for all the experiments with neural networks.

5.3.2 Decision Trees

All three classes of experiments were conducted by creating data sets that had the loop unroll gain nominal value as the feature value to be predicted. The loop unroll gain nominal value is derived from the loop unroll gain numerical value by labeling it as "yes" if unrolling is beneficial and "no" if not. The decision tree results were evaluated by considering whether the loop unrolling gain labels were predicted correctly.

Experiments were conducted for loop data with feature sets A and B by creating decision trees with and without pruning, as described in section 4.2.

Benchmark	Input Data Size	Iterations
BT	12×12×12	50
	24×24×24	50
	36×36×36	50
	48×48×48	50
	64×64×64	50
	84×84×84	50
	102×102×102	50
CG	1400	15
	7000	15
	14000	15
	40000	15
	60000	15
	75000	15
FT	64×64×64	16
	128×128×32	16
	256×256×128	16
	512×256×256	16
LU	12×12×12	50
	33×33×33	50
	48×48×48	50
	64×64×64	50
	84×84×84	50
	102×102×102	50
MG	16×16×16	20
	32×32×32	20
	64×64×64	20
	128×128×128	20
	256×256×256	20
SP	12×12×12	100
	24×24×24	100
	36×36×36	100
	48×48×48	100
	64×64×64	100
	84×84×84	100
	102×102×102	100

Table 5.1: Input data sizes for OpenMP NAS benchmarks.

Number of threads used.
Nesting depth of the loop.
Trip count of the loop.
Number of statements in the loop.
Number of floating point additions and subtractions.
Number of floating point multiplications and divisions.
Number of floating point exponentiations.
Number of floating point variables.

Table 5.2: Feature set A: Loop characterization

Number of threads used.
Nesting depth of the loop.
Trip count of the loop.
Number of statements in the loop.
Number of floating point operations.
Number of floating point variables.

Table 5.3: Feature set B: Loop characterization

Chapter 6

Results and Analysis

In this section, the accuracy of Neural Networks (NN) and Decision Trees (DT) for predicting the effectiveness of loop unrolling is presented. The "raw accuracy" metric defined in equation 5.1 is used to evaluate the performance of both NN and DT. In addition, the performance of NN using the 95% confidence interval metric is also evaluated. The DT experiments were conducted with and without pruning of the derived tree. Because the results of pruning the decision tree were decidedly poorer than using an unpruned tree, only the results with unpruned trees are reported for DT in this section.

The results for both NN and DT are reported using feature sets A and B for characterizing loops. In addition, results for neural networks are reported in the prescence and abscence of validation sets, in order to test the sensitivity of results to the degree of availability of information.

There are four sets of experiments that are performed and evaluated for Neural Networks:

- Case 1: NN with feature set A and validation set
- Case 2: NN with feature set A and without validation set
- Case 3: NN with feature set B and validation set
- Case 4: NN with feature set B and without validation set

There are two sets of experiments that are performed and evaluated for Decision Trees:

- Case 1: DT with feature set A
- Case 2: DT with feature set B

Results are reported for the interpolation class of experiments for both NN and DT, with two kinds of partitioning of available data across threads within a single benchmark:

- Interpolation case v1: Given loop data available about 2, 4 and 8 threads for training, predict unrolling performance for 6 threads as testing.
- Interpolation case v2: Given loop data available about 2, 6 and 8 threads for training, predict unrolling performance for 4 threads as testing.

Results are reported for the extrapolation class of experiments for both NN and DT, with two kinds of partitioning of data available across threads within a single benchmark:

- Extrapolation case v1: Given loop data available about 2 and 4 threads for training, predict unrolling performance for 6 threads as testing.
- Extrapolation case v2: Given loop data available about 2, 4 and 6 threads for training, predict unrolling performance for 8 threads as testing.

What we are trying to find out with the interpolation and extrapolation class of experiments is how well NN and DT are able to predict unrolling effectiveness for the thread values it has not been presented based on the ones that are presented.

Results are reported for the benchmark-to-benchmark class of experiments for both NN and DT, by utilizing across all 2, 4, 6 and 8 threads for each benchmark.

6.1 Neural Networks(NN)

6.1.1 Case 1: NN with feature set A and with validation set

Interpolation

Figure 6.1 shows the raw accuracy for predicting whether loop unrolling is beneficial for interpolation across threads within the same benchmark. The average raw accuracy for all the cases is 70.74%. The average raw accuracy for all v1 cases is 71.30%. The average raw accuracy for all v2 cases is 70.19%.

The loop unrolling benefit prediction accuracy for interpolation is very similar for both v1 and v2 cases, for all benchmarks.

It can thus be concluded that neural networks is quite accurate for interpolating loop unrolling performance across threads within the same benchmark.

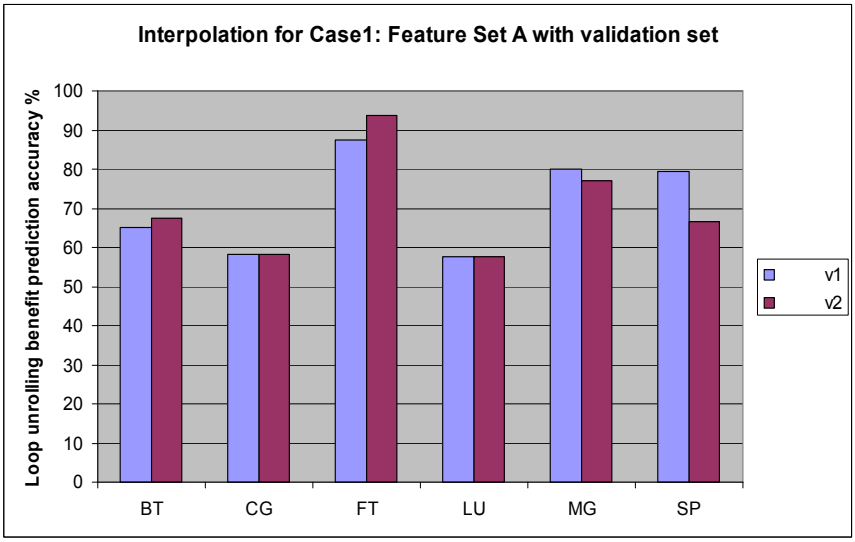


Figure 6.1: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Interpolation Case 1

Extrapolation

Figure 6.2 shows the raw accuracy for predicting whether loop unrolling is beneficial for extrapolation across threads within the same benchmark. The average raw accuracy for all the cases is 63.23%. The average raw accuracy for all v1 cases is 60.81%. The average

raw accuracy for all v2 cases is 65.66%.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases, except in the case of BT. The accuracy for extrapolation is considerably less than for interpolation.

It can thus be concluded that neural networks is not as accurate for extrapolating loop unrolling performance across threads within the same benchmark.

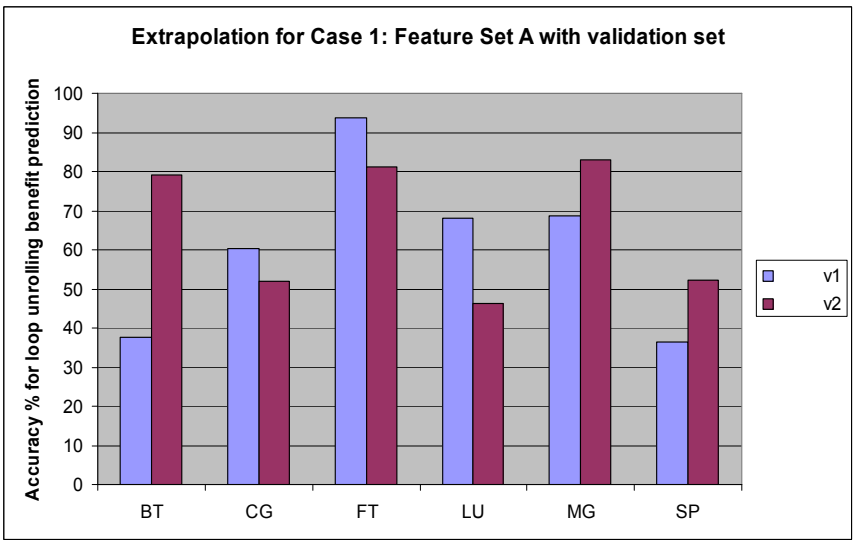


Figure 6.2: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Extrapolation Case 1

Benchmark-to-benchmark prediction

Figure 6.3 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose loops are used to predict the performance of the remaining benchmarks. The average raw accuracy for all the cases is 60.58%.

Figure 6.4 shows the 95% confidence accuracy for predicting whether loop unrolling is beneficial across benchmarks. The average 95% confidence accuracy for all the cases is 57.6%. The average 95% confidence accuracy for confident points is 61.39%. The 95% confidence accuracy for loop unrolling behavior prediction for confident points is very close to the raw accuracy number.

The NN is able to capture the behavior of the self loops of each benchmark ade-

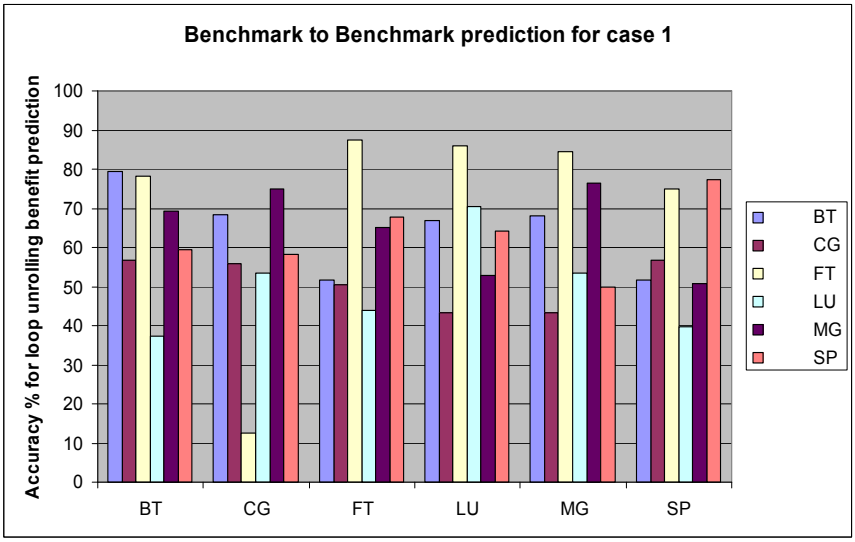


Figure 6.3: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 1

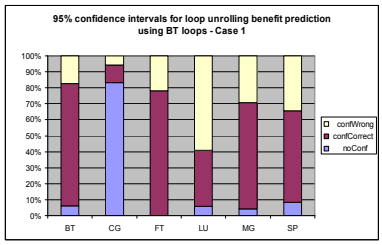
quately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

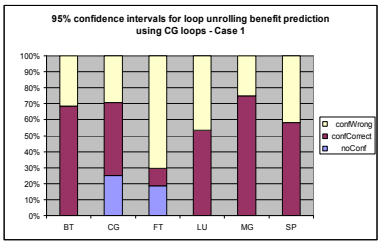
In general, the 60.58% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance.

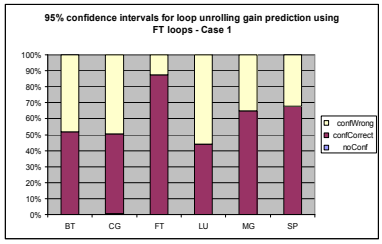
6.1.2 Case 2: NN with feature set A and without validation set

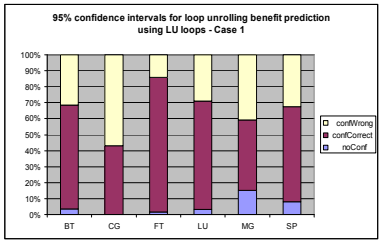
Interpolation

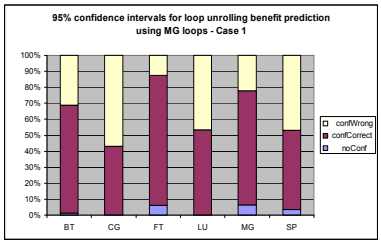
Figure 6.5 shows the raw accuracy for predicting whether loop unrolling is beneficial for interpolation across threads within the same benchmark. The average raw accuracy for all the cases is 71.49%. The average raw accuracy for all v1 cases is 74.25%. The average raw accuracy for all v2 cases is 68.73%.











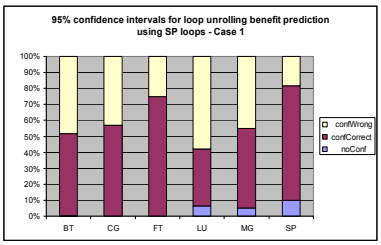


Figure 6.4: 95% confidence accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 1

The loop unrolling benefit prediction accuracy for interpolation is very similar for both v1 and v2 cases, for all benchmarks, and is marginally better than case 1 using the validation set.

It can thus be concluded that neural networks is quite accurate for interpolating loop unrolling performance across threads within the same benchmark.

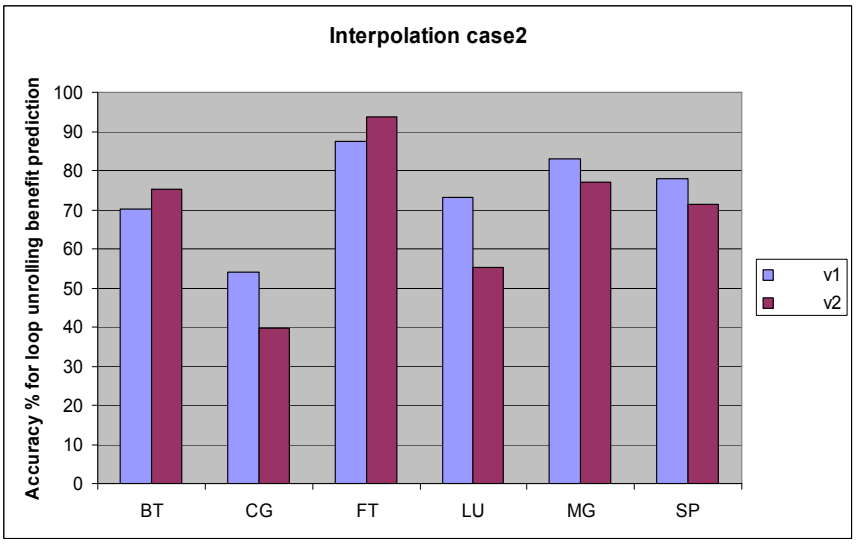


Figure 6.5: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Interpolation Case 2

Extrapolation

Figure 6.6 shows the raw accuracy for predicting whether loop unrolling is beneficial for extrapolation across threads within the same benchmark. The average raw accuracy for all the cases is 66.77%. The average raw accuracy for all v1 cases is 68.80%. The average raw accuracy for all v2 cases is 64.75%.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases, , and is significantly better than case 1 using the validation set. The accuracy for extrapolation is considerably less than for interpolation.

Benchmark-to-benchmark prediction

Figure 6.7 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose

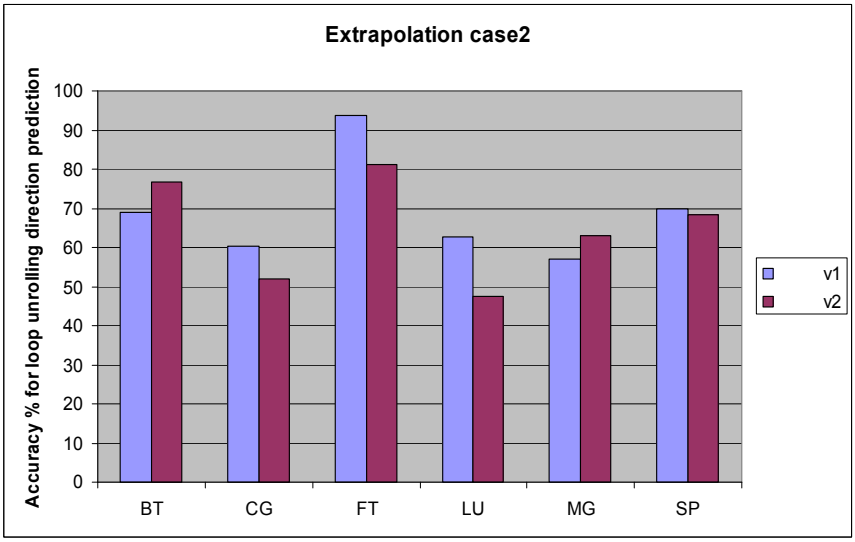


Figure 6.6: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Extrapolation Case 2

loops are used to predict the performance of the remaining benchmarks. The average raw accuracy for all the cases is 62.13%.

Figure 6.8 shows the 95% confident accuracy for predicting whether loop unrolling is beneficial across benchmarks. The average 95% confidence for all the cases is 59.98%. The average 95% confidence accuracy for confident points is 62.51%. The 95% confidence accuracy for loop unrolling behavior prediction for confident points is very close to the raw accuracy number.

The NN is able to capture the behavior of the self loops of each benchmark adequately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

In general, the 62.13% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance. Using all the available data has improved the accuracy as compared to case 1 with validation set.

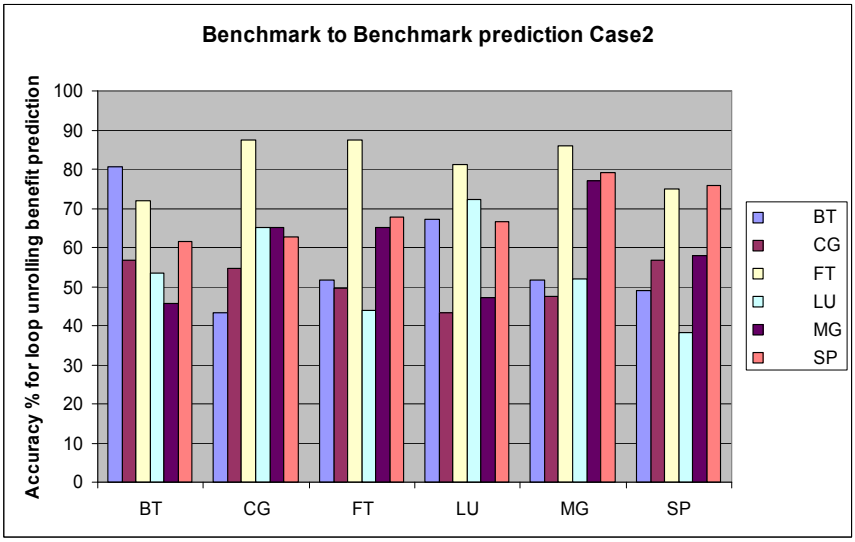


Figure 6.7: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 2

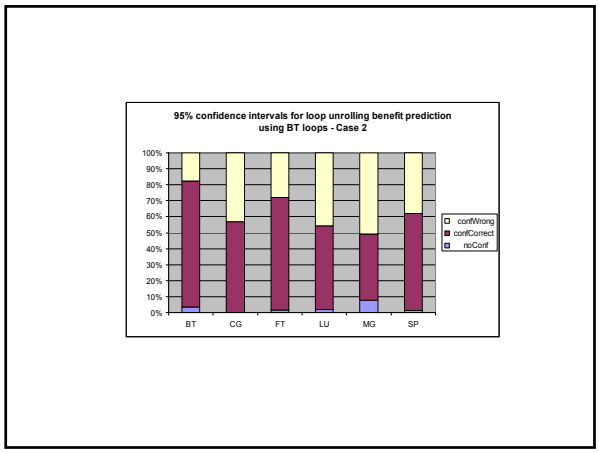
6.1.3 Case 3: NN with feature set B and with validation set

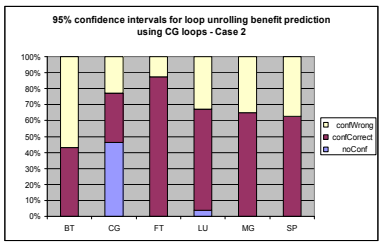
Interpolation

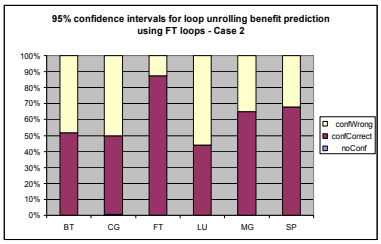
Figure 6.9 shows the raw accuracy for predicting whether loop unrolling is beneficial for interpolation across threads within the same benchmark. The average accuracy for all the cases is 67.5%. The average raw accuracy for all v1 cases is 72.36%. The average raw accuracy for all v2 cases is 62.64%.

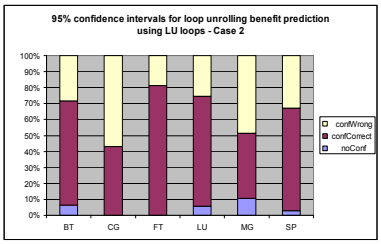
The loop unrolling benefit prediction accuracy for interpolation is very similar for both v1 and v2 cases, for all benchmarks. It is however perceptibly less than the one for case 1 with feature set A with validation set, and case 2 with feature set A and without validation set.

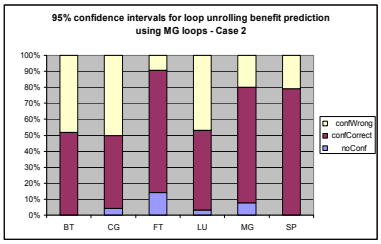
It can thus be concluded that neural networks is quite accurate for interpolating loop unrolling performance across threads within the same benchmark, and characterizing floating point operations individually(feature set A, cases 1 and 2) instead of in aggregation(feature set B, case 3 herein)is significantly better for interpolation.











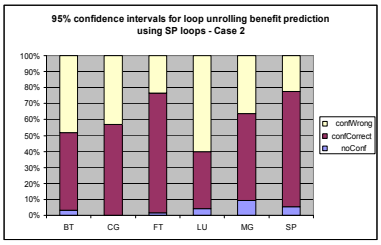


Figure 6.8: 95% confidence accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 2

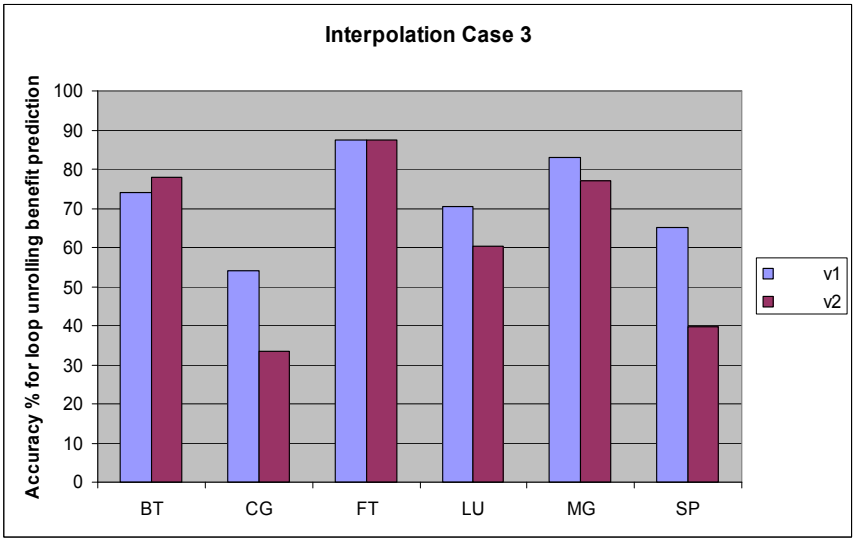


Figure 6.9: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Interpolation Case 3

Extrapolation

Figure 6.10 shows the raw accuracy for predicting whether loop unrolling is beneficial for extrapolation across threads within the same benchmark. The average accuracy for all the cases is 64.69%. The average raw accuracy for all v1 cases is 68.16%. The average raw accuracy for all v2 cases is 61.23%.

We observe that the loop unrolling benefit prediction performance is similar to cases 1 and 2.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases. The accuracy for extrapolation is considerably less than for interpolation, and is less than that using feature set A (case 2).

It can thus be concluded that neural networks is not as accurate for extrapolating loop unrolling performance across threads within the same benchmark.

Benchmark-to-benchmark prediction

Figure 6.11 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose loops are used to predict the performance of the remaining benchmarks. The average raw accuracy for all the cases is 59.21%.

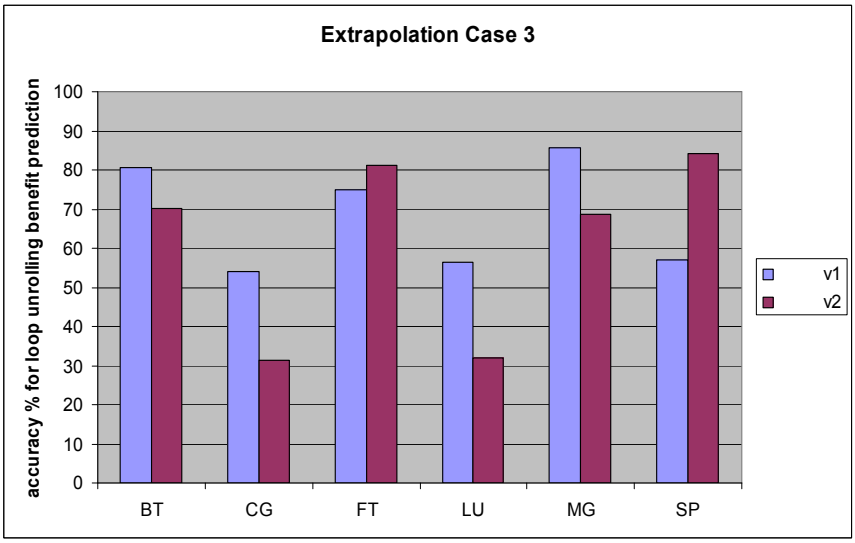


Figure 6.10: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Extrapolation Case 3

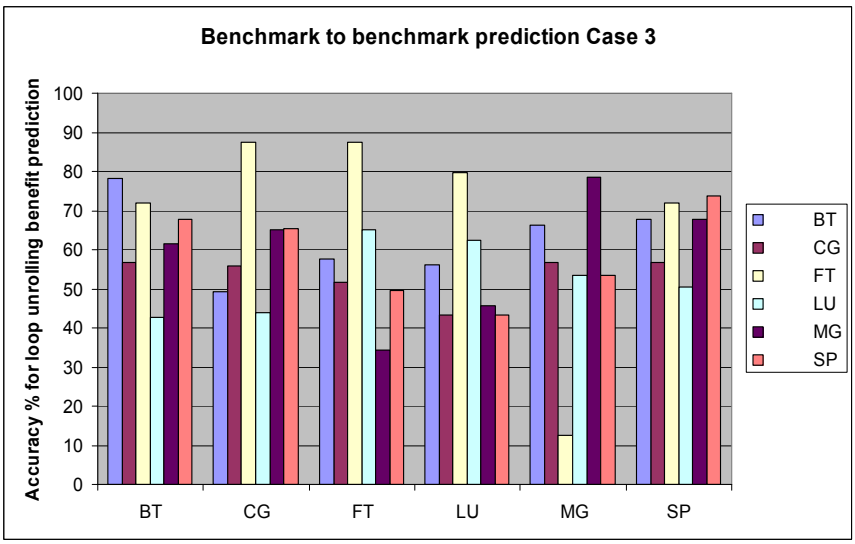


Figure 6.11: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 3

Figure 6.12 shows the 95% confidence accuracy for predicting whether loop unrolling is beneficial across benchmarks. The average 95% confidence accuracy for all the cases is 59.98%. The average 95% confidence accuracy for confident points is 62.51%. The 95% confidence accuracy for loop unrolling behavior prediction for confident points is roughly comparable to the raw accuracy number.

The NN is able to capture the behavior of the self loops of each benchmark adequately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

In general, the 59.21% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance and is slightly less than that for cases 1 and 2 that use feature set A.

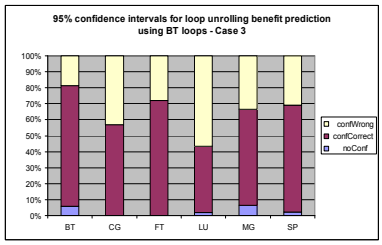
6.1.4 Case 4: NN with feature set B and without validation set

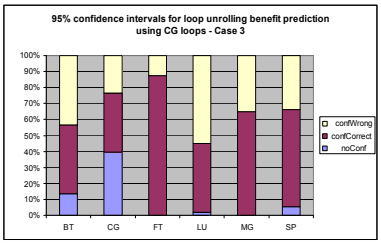
Interpolation

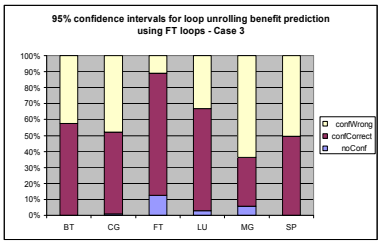
Figure 6.13 shows the raw accuracy for predicting whether loop unrolling is beneficial for interpolation across threads within the same benchmark. The average accuracy for all the cases is 69.72%. The average raw accuracy for all v1 cases is 71.24%. The average raw accuracy for all v2 cases is 68.21%.

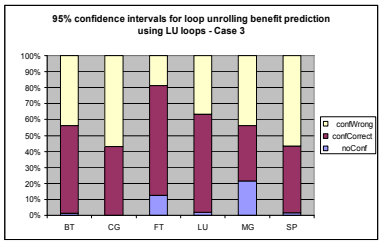
The loop unrolling benefit prediction accuracy for interpolation is very similar for both v1 and v2 cases, for all benchmarks. It is however perceptibly more than the one for case 3 with validation set, and less than cases 1 and 2 with feature set A.

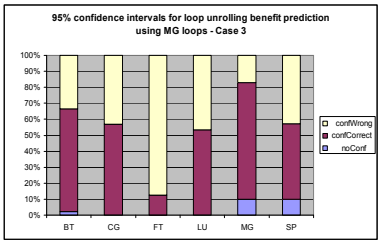
It can thus be concluded that neural networks is quite accurate for interpolating loop unrolling performance across threads within the same benchmark, and characterizing floating point operations individually(feature set A, cases 1 and 2) instead of in aggregation(feature set B, case 3 herein)is significantly better for interpolation.











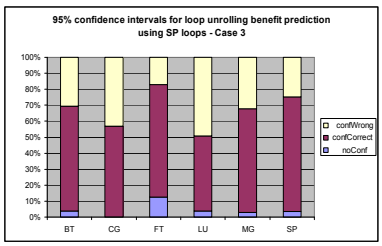


Figure 6.12: 95% confidence accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 3

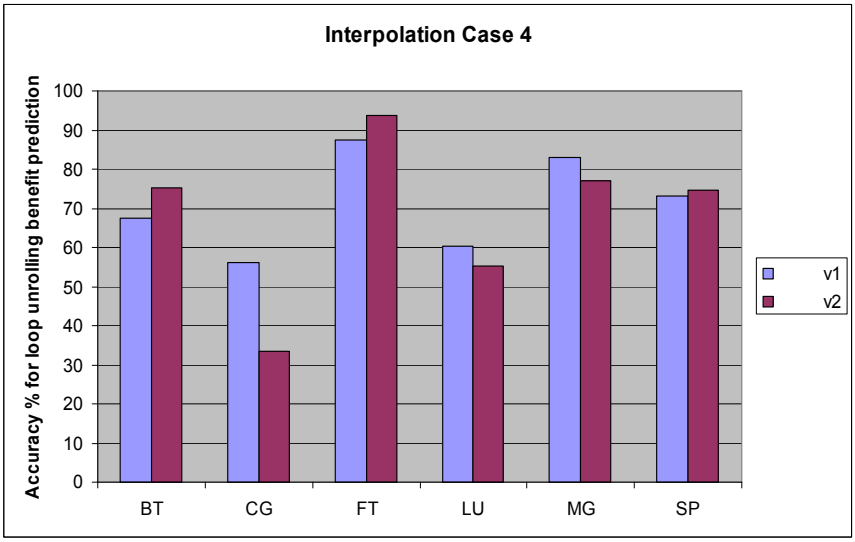


Figure 6.13: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Interpolation Case 4

Extrapolation

Figure 6.14 shows the raw accuracy for predicting whether loop unrolling is beneficial for extrapolation across threads within the same benchmark. The average accuracy for all the cases is 63.54% The average raw accuracy for all v1 cases is 65.15%. The average raw accuracy for all v2 cases is 61.92%.

The loop unrolling benefit prediction performance is similar to cases 1 and 2.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases. The accuracy for extrapolation is considerably less than for interpolation, and is less than that using validation (case 2) set and feature set A (cases 1 and 2, with and without validation set, respectively).

It can thus be concluded that neural networks is not as accurate for extrapolating loop unrolling performance across threads within the same benchmark.

Benchmark-to-benchmark prediction

Figure 6.15 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose loops are used to predict the performance of the remaining benchmarks. The average accuracy for all the cases is 56.05%.

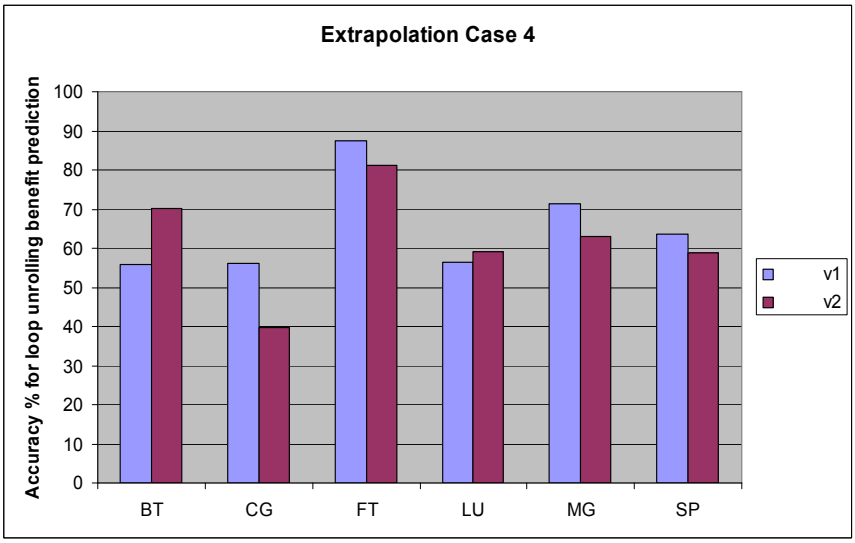


Figure 6.14: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Extrapolation Case 4

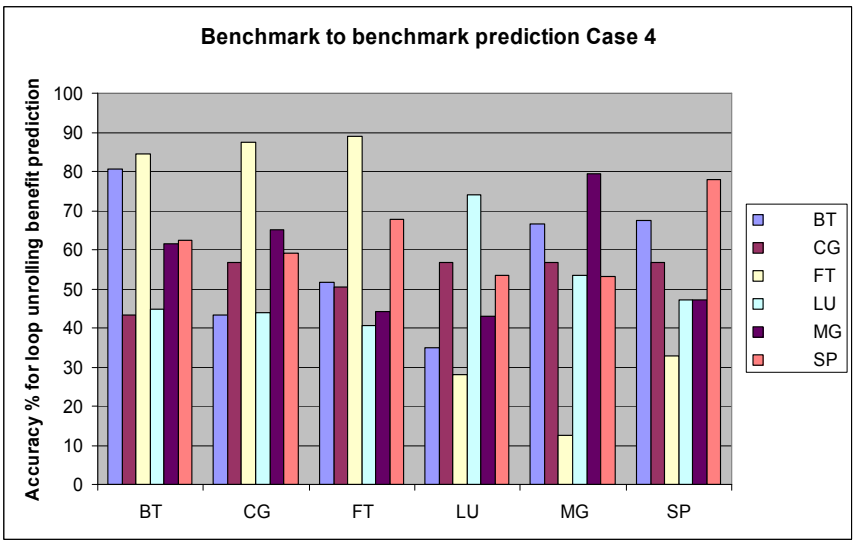


Figure 6.15: Raw accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 4

Figure 6.16 shows the 95% confident accuracy for predicting whether loop unrolling is beneficial across benchmarks. The average 95% confidence for all the cases is 54.27%. The average 95% confidence accuracy for confident points is 56.51%. The 95% confidence accuracy for loop unrolling behavior prediction for confident points is roughly comparable to the raw accuracy number.

The NN is able to capture the behavior of the self loops of each benchmark adequately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

In general, the 56.05% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance and is significantly less than that for case 3 that uses a validation set, and cases 1 and 2 that use feature set A.

6.2 Decision Trees

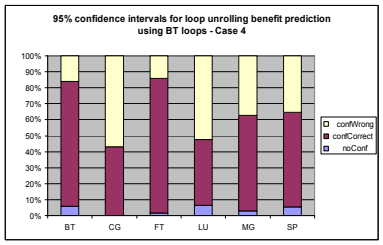
6.2.1 Case 1: Feature Set A

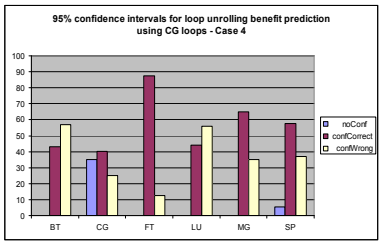
Interpolation

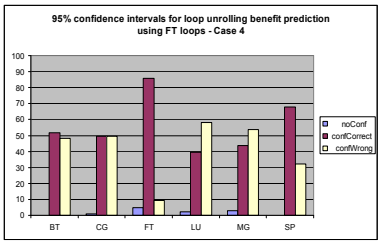
Figure 6.17 shows the raw accuracy for predicting whether loop unrolling is beneficial, by interpolation across threads within the same benchmark. The average raw accuracy for all the cases is 70.33%. The average accuracy for all v1 cases is 70.38%. The average accuracy for all v2 cases is 70.28%.

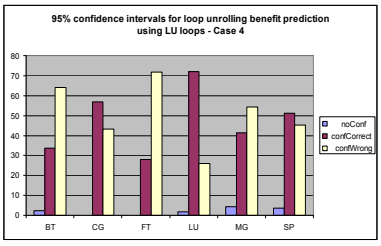
The loop unrolling benefit prediction accuracy for interpolation is very similar for both v1 and v2 cases, for all benchmarks.

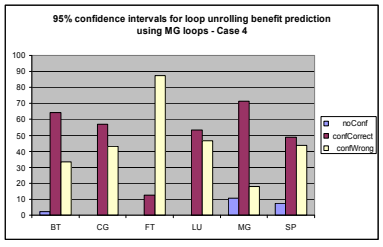
It can thus be concluded that decision tree is quite accurate for interpolating loop unrolling performance across threads within the same benchmark.











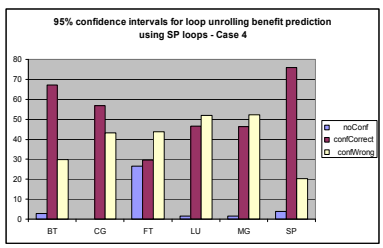


Figure 6.16: 95% confidence accuracy for Loop Unrolling benefit Prediction % using Neural Networks: Benchmark-to-benchmark Case 4

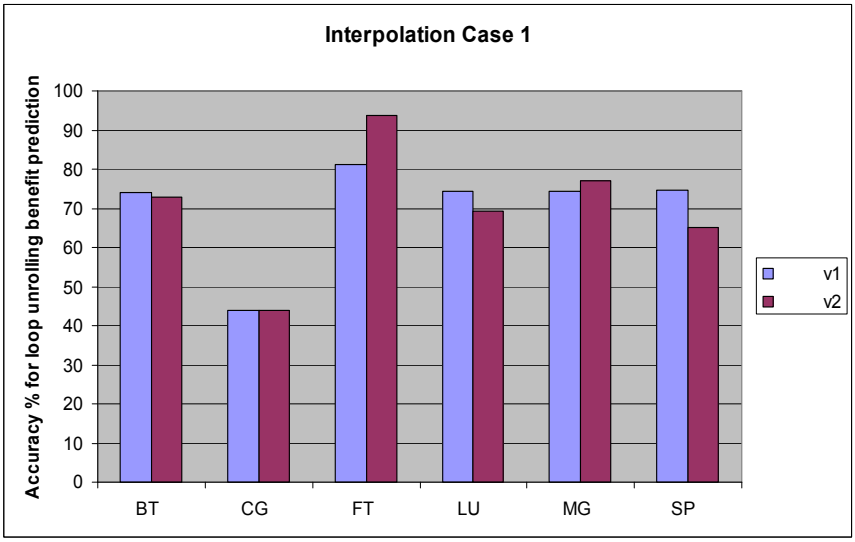


Figure 6.17: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Interpolation Case 1

Extrapolation

Figure 6.18 shows the raw accuracy for predicting whether loop unrolling is beneficial, by extrapolation across threads. The average raw accuracy for all the cases is 70.67%. The average raw accuracy for all v1 cases is 71.49%. The average raw accuracy for all v2 cases is 69.84%.

The accuracy for both v1 and v2 is similar, and the net accuracy is similar to interpolation.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases. The accuracy for extrapolation is considerably less than for interpolation.

It can thus be concluded that decision tree is as accurate for extrapolating loop unrolling performance across threads within the same benchmark as interpolation.

Benchmark-to-benchmark prediction

Figure 6.19 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose loops are used to predict the performance of the remaining benchmarks. The average accuracy for all the cases is 56.80%.

The NN is able to capture the behavior of the self loops of each benchmark ade-

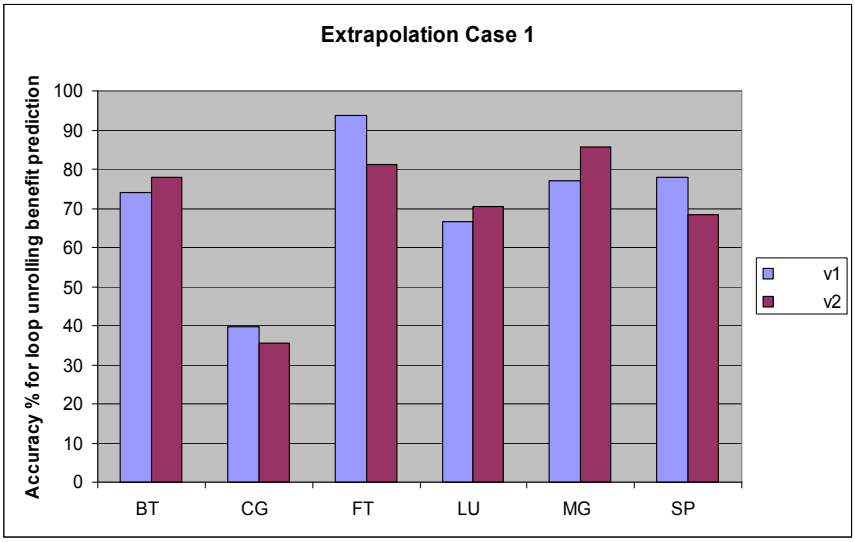


Figure 6.18: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Extrapolation Case 1

quately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

In general, the 56.8% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance.

6.2.2 Case 2: Feature Set B

Interpolation

Figure 6.20 shows the raw accuracy for predicting whether loop unrolling is beneficial for interpolation across threads within the same benchmark. The average accuracy for all the cases is 70.17%. The average accuracy for all v1 cases is 71.02%. The average accuracy for all v2 cases is 69.32%.

The loop unrolling benefit prediction accuracy for interpolation is very similar for

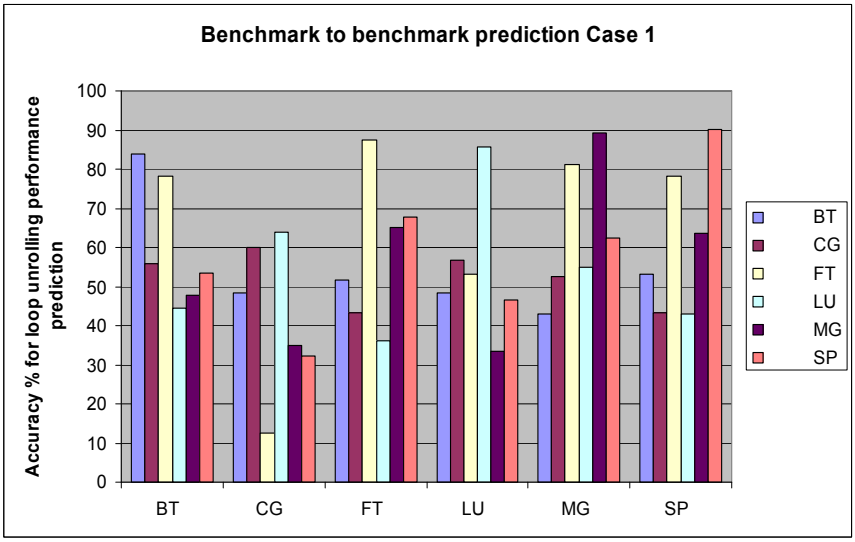


Figure 6.19: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Benchmark-to-benchmark Case 1

both v1 and v2 cases, for all benchmarks, and is comparable to using feature set A, as in case 1.

It can thus be concluded that decision tree is quite accurate for interpolating loop unrolling performance across threads within the same benchmark.

Extrapolation

Figure 6.21 shows the raw accuracy for predicting whether loop unrolling is beneficial for extrapolation across threads within the same benchmark. The average accuracy for all the cases is 70.87%. The average accuracy for all v1 cases is 72.56%. The average accuracy for all v2 cases is 69.18%.

The accuracy for both v1 and v2 is similar, and the net accuracy is similar to interpolation.

The loop unrolling benefit prediction accuracy for extrapolation is comparable for v1 and v2 cases. The accuracy for extrapolation is quite similar to interpolation, and is similar to case 1 using feature set A.

It can thus be concluded that decision tree is as accurate for extrapolating loop unrolling performance across threads within the same benchmark as interpolation.

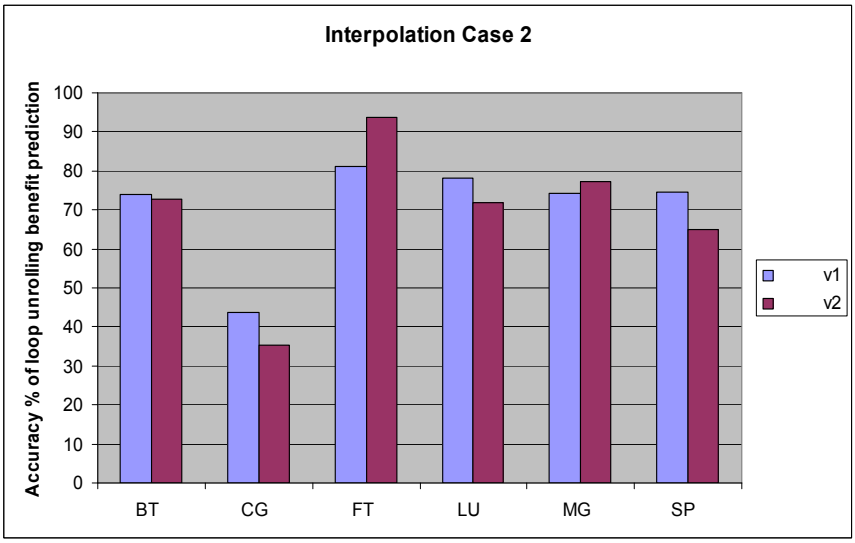


Figure 6.20: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Interpolation Case 2

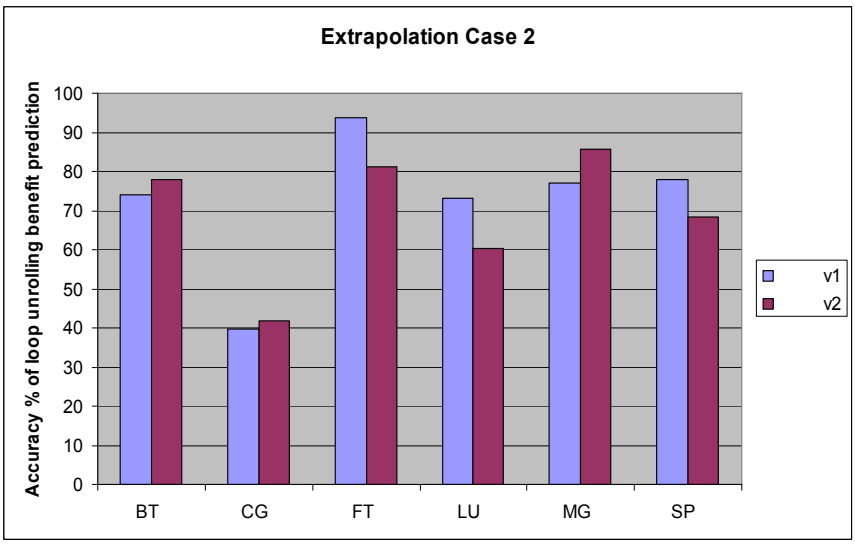


Figure 6.21: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Extrapolation Case 2

Benchmark-to-benchmark prediction

Figure 6.19 shows the raw accuracy for predicting whether loop unrolling is beneficial across benchmarks. The benchmark noted in the x-axis of the graph is the one whose loops are used to predict the performance of the remaining benchmarks. The average accuracy for all the cases is 53.68%.

The NN is able to capture the behavior of the self loops of each benchmark adequately, except in the case of CG. This may be because CG tests unstructured computations and communications in a randomly generated sparse matrix and consequently, the data access patterns of CG may lead to lot of data cache misses. Data access patterns are not captured by our loop feature set. The unrolling behavior of loops in LU is not captured adequately by the loops of any other benchmark. This may again be because data access patterns of LU are drastically different from the remaining benchmarks and are not captured by the loop features that are employed.

In general, the 53.68% raw accuracy maybe because of the inadequacy of source code loop features to capture adequately the parameters that impact loop unrolling performance, and is considerably less than that for case 1 using feature set A.

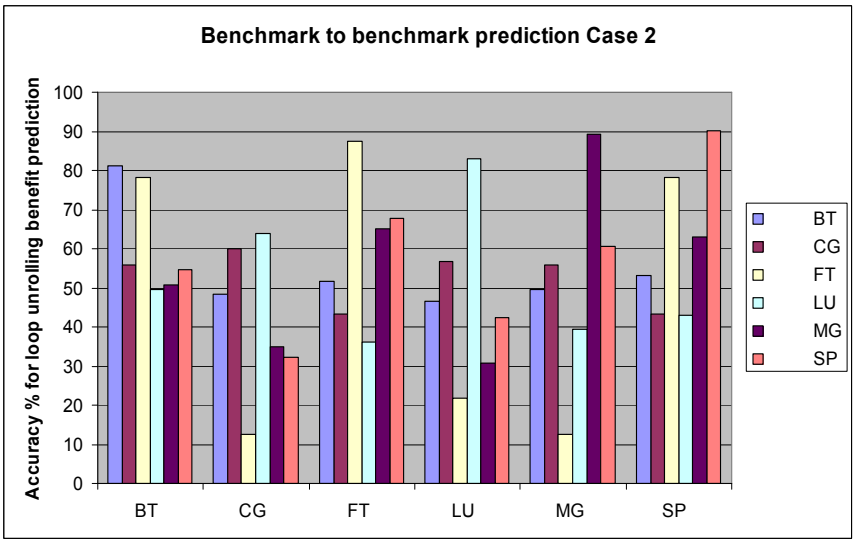


Figure 6.22: Raw accuracy for Loop Unrolling benefit Prediction % using Decision Trees: Benchmark-to-benchmark Case 2

6.3 Discussion

Using our loop characterization strategy, the loop unrolling behavior can be extrapolated and interpolated across the number of threads within NAS benchmarks to a raw accuracy of roughly 70% using decision trees. Using neural networks, the raw interpolation accuracy of 71.49% in the best case is significantly higher than the raw extrapolation accuracy of 66.77% in the best case.

However for predicting loop unrolling behavior across NAS benchmarks, the best raw accuracy is only 62.13% using neural networks and 56.8% using decision trees.

The loop unrolling behavior of CG across the number of threads in general is not captured well even by itself. The behavior of LU's loops is not captured by that of any other benchmark but itself. BT, FT, MG and SP loop unrolling behavior is adequately captured by their own loops using machine learning. Using the validation set in neural networks during training overall gives marginally less accurate results.

The conclusion is that using simple loop features derived from source code, machine learning does not adequately capture the loop unrolling behavior across applications. This may be because interaction of loop unrolling with memory hierarchy, data layout and data access patterns which varies with program phase and may significantly impact the loop unrolling behavior are not captured by our loop features. Thus, loops that happen to have identical to near identical signatures in our scheme show varying unrolling behavior resulting in a much lower prediction performance in shared multiprocessor environment.

Chapter 7

Related Work

Monsifrot et al. [11] use a decision tree to determine whether to unroll a loop based on loop features statically extracted from source code, similar to what has been attempted in this thesis. However the work done in in this thesis studies the problem in the context of parallel programs running on shared memory multiprocessors whereas [11] focusses on sequential programs running on uniprocessors.

Mark et al. [18] used supervised learning techniques such as near neighbor (NN) classification and Support Vector Machines (SVM) to assist the compiler in deciding the optimal unroll factor for a given loop. Our work only concentrates on deciding whether to unroll a loop or not, and assumes that the performance tuners do not have the ability to modify the compiler.

Machine learning techniques have also been used to improve branch prediction acccuracy. Calder et al. used neural networks and decision trees to perform static branch prediction at compile time [2, 3]. Vintan and Iridon have suggested learning vector quantization and neural networks for dynamic branch prediction [10, 5]. Jiménez et al. use perceptrons for dynamic branch prediction [8, 7].

Chapter 8

Conclusions

To the best of our knowledge, this thesis is the first to test whether machine learning techniques can be applied for predicting loop unrolling effectiveness on parallel shared memory programs. The experiments in this thesis showed that Artificial Neural Network (ANN) and Decision Tree (DT) can be used to some degree of accuracy in predicting the effectiveness of loop unrolling. They are more accurate when they predict the effectiveness across different number of threads for the same application, but less accurate when they predict the effectiveness across different applications. Comparing the ANN and DT, ANN performs slightly better than DT for predicting across applications, and predicting for the same benchmark through interploation. For predicting for the same benchmark through extrapolation, DT performs better than ANN. Characterizing loops by considering floating point operations individually rather than in aggregation improves accuracy marginally in most of the cases.

Bibliography

[1] Francine Berman and Richard Wolski. Scheduling from the perspective of the application. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*, page 100, Washington, DC, USA, 1996. IEEE Computer Society.

[2] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[3] Brad Calder, Dirk Grunwald, Donald C. Lindsay, James Martin, Michael Mozer, and Benjamin G. Zorn. Corpus-based static branch prediction. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–92, 1995.

[4] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[5] Colin Egan, Gordon Steven, Patrick Quick, Rubén Anguera, Fleur Steven, and Lucian Vintan. Two-level branch prediction using neural networks. *J. Syst. Archit.*, 49(12-15):557–570, 2003.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan-Kaufmann Publishers, Inc., 2nd edition, 1996.

[7] Daniel A. Jiménez. Improved latency and accuracy for neural branch prediction. *ACM Trans. Comput. Syst.*, 23(2):197–218, 2005.

[8] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[9] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report 011, NASA, 1999.

[10] L.N.Vintan and M.Iridon. Towards a high-performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 868–873. IEEE Press, 1999.

[11] Antoine Monsifrot, Francois Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, London, UK, 2002. Springer-Verlag.

[12] Dimitrios S. Nikolopoulos, Eduard Ayguadé, and Constantine D. Polychronopoulos. Runtime vs. Manual Data Distribution for Architecture-Agnostic Shared-Memory Programming Models. *Int. J. Parallel Program.*, 30(4):225–255, 2002.

[13] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 415–427, London, UK, 2000. Springer-Verlag.

[14] Lutz Prechelt. PROBEN1 — A set of benchmarks and benchmarking rules for neural network training algorithms. Technical Report 21/94, Fakultät für Informatik, Universität Karlsruhe, D-76128 Karlsruhe, Germany, September 1994. Anonymous FTP: /pub/papers/techreports/1994/1994-21.ps.Z on ftp.ira.uka.de.

[15] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 172, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst.*, 18(1):175–187, 2001.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, et al., editors, *Parallel Distributed Processing: Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, 1987.

[18] Mark Stephenson and Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[20] R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.

[21] Ian H. Witten and Eibe Frank. Data mining: practical machine learning tools and techniques with Java implementations. *SIGMOD Rec.*, 31(1):76–77, 2002.